AD tool usability and OpenAD

- 4 motivations for automatic differentiation
- at first glance it looks like a compiler to the user
- after a while it seems to get a bit more complicated
- what do we do with OpenAD

4 motivations for automatic differentiation

we have a some model given as a (large) program

1. pretend to know nothing about the program and take finite differences of an oracle? - perhaps not.
2. get machine precision derivatives (avoid approximation vs. rounding problem)
3. the reverse mode (program reversal) yields "cheap gradients"
4. if the program is large, so is the adjoint, so is the effort to do it manually ... and it is easy to get wrong but hard to debug

get a tool to do it "automatically"

Looks like a compiler to me

- a simple user setup: the entire model code with the top level routine
  `subroutine foo(x,y)`
  input `x` and output `y`.
- feed this to a tool that
  - parses the input code
  - for each construct found in the input create a new construct that does the "derivative computation"
  - integrate all pieces into a new program (or may be even an executable)
    e.g. for `subroutine foo_bar(x_bar,y_bar)` where,
    `x_bar`$=\frac{\partial y}{\partial x}$
- run `foo_bar` and be done
- but may be it is rather like run `foo_bar` ... ... wait ... wait some more ... wait even longer ... not done yet ... ... ran out of memory
- a simplistic approach is not enough - how about "activity analysis"?

still looks like a compiler to me

- assume the model code with the top level routine

  `subroutine foo(` $\overbrace{\texttt{y,q,r,s}}^{\texttt{inputs}}$`,` $\overbrace{\texttt{x,t,u,v}}^{\texttt{outputs}}$ `)`

  x, **y**, and *passive parameters* `q, r, s, t, u, v`.
- we are only interested in derivatives involving *active* variables `x` and `y`
- designate **x** is *independent* and **y** as *dependent*
- use specialized compiler-style data-flow analysis to generate `foo_bar` only for computations that depend on `x` and also impact `y`.
- `foo_bar` takes less time
- now try it again run `foo_bar` ... wait ... wait some more ... ...
  ... hmm, out of memory - again ☹
- Why memory? Cheap gradients cost memory!

a little reminder

`foo` contains:

$a = \alpha(x)$

$b = \beta(a)$

$\vdots$

$y = \gamma(b)$

`foo_bar` code has:

$\bar{b} = \bar{b} + \bar{y} \cdot \frac{\partial \gamma}{\partial b}$ ← pop

$\bar{y} = 0$

$\vdots$

$\bar{a} = \bar{a} + \bar{b} \cdot \frac{\partial \beta}{\partial a}$ ← pop

$\bar{b} = 0$

$\bar{x} = \bar{x} + \bar{a} \cdot \frac{\partial \alpha}{\partial x}$ ← pop

$\bar{a} = 0$

so we may tape the needed partials:

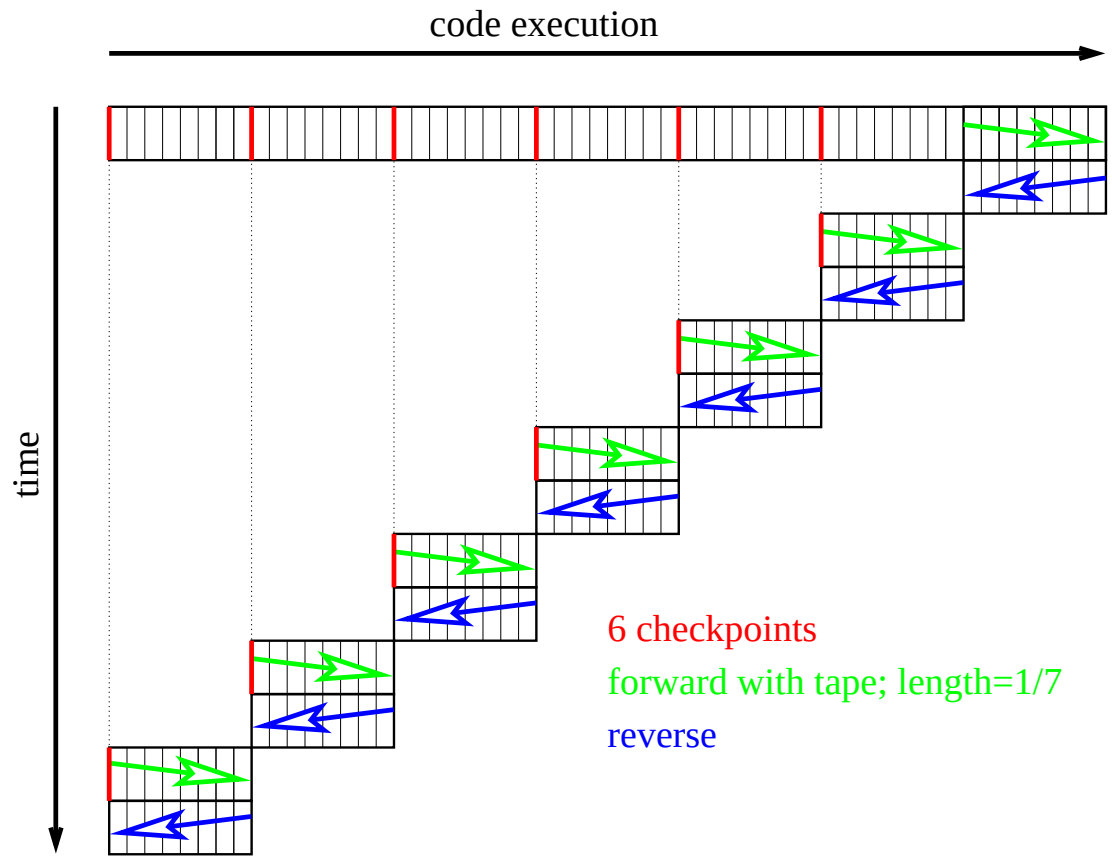
$a = \alpha(x)$; `push` $\frac{\partial \alpha}{\partial x}$

$b = \beta(a)$; `push` $\frac{\partial \beta}{\partial a}$

$\vdots$

$y = \gamma(b)$; `push` $\frac{\partial \gamma}{\partial b}$

trade memory consumption for recomputation



- control checkpoint locations via pragmas
- determine checkpoint contents using compiler-like side effect analysis
- hierarchy of checkpoints
- checkpoint size vs. tape size reductions
- how should one control irregular checkpointing/reversal schemes?

... it is becoming less compiler - like ...

obfuscating code → confused tool

- usually the first victim is activity analysis.
- example: write intermediate state to a file, later read that state from the file (and may be throw in constructed file names).
  - conventional analysis looses track
  - wrap file i/o into subroutines and present "analyzable" code to the tool
- black box routines
- type recasting (use of `EQUIVALENCE`)
- extensive use of pointer arithmetic (in C/C++)

more manual intervention

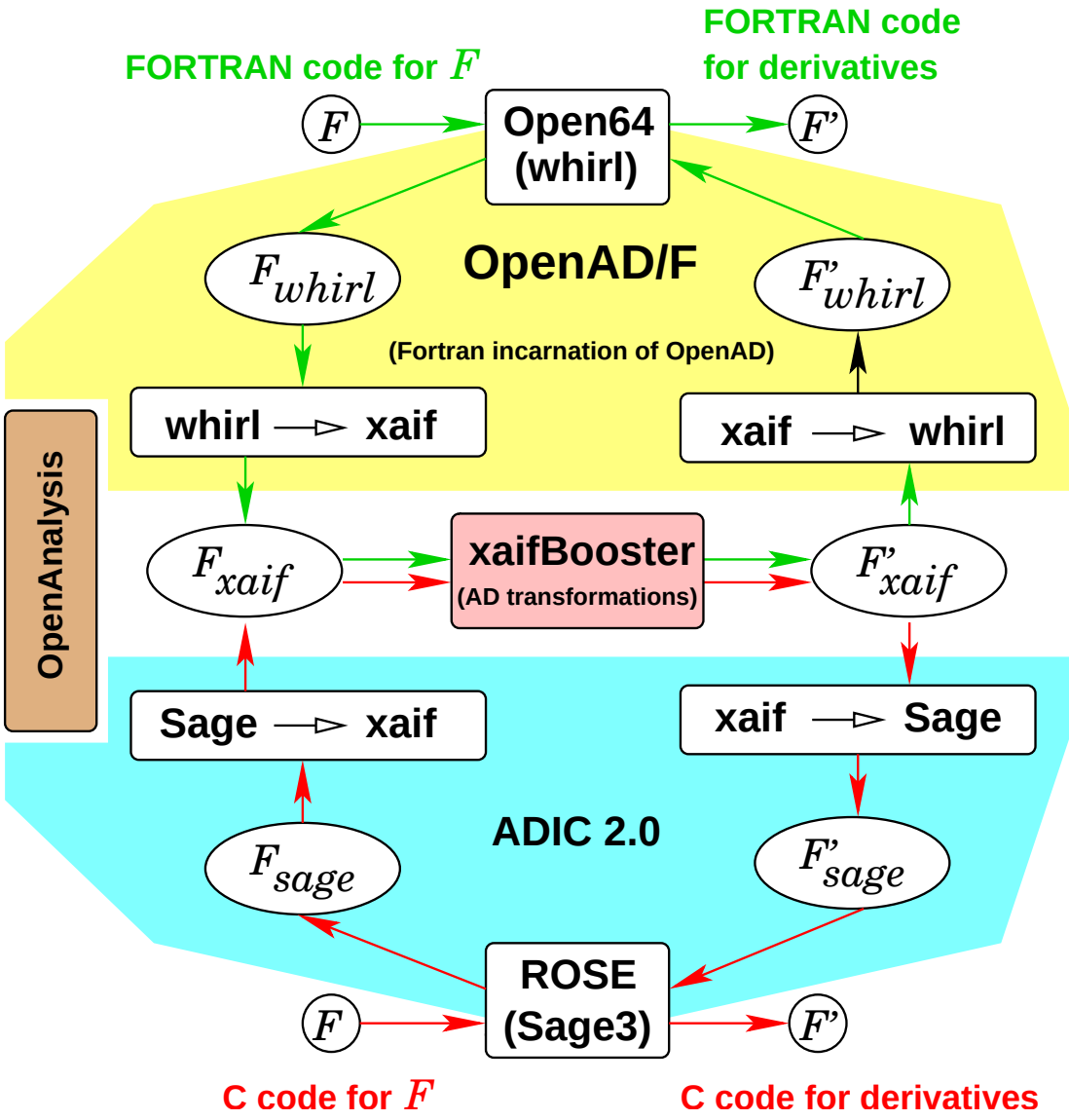
- have lots of extras for environment setup/output/...
- show the AD tool only parts of the code
  - reduce conservative (over)estimates, e.g. overestimate of active variable set
  - avoid confusing the analysis with irrelevant/difficult code
  - cut down on analysis time
  - have to manually ensure hidden parts fit seamlessly!
- self adjoint subroutines
  - hide from tool
  - manually adjoin via wrapping code (unless there is a generic interface)
- parallel processing
  - possibly hide data exchange / execution control
  - manually adjoin via wrapping code (tools are getting better)

.... all of the above is distinctly not compiler-like. ...

OpenAD (ACTS) etc.

some goals:

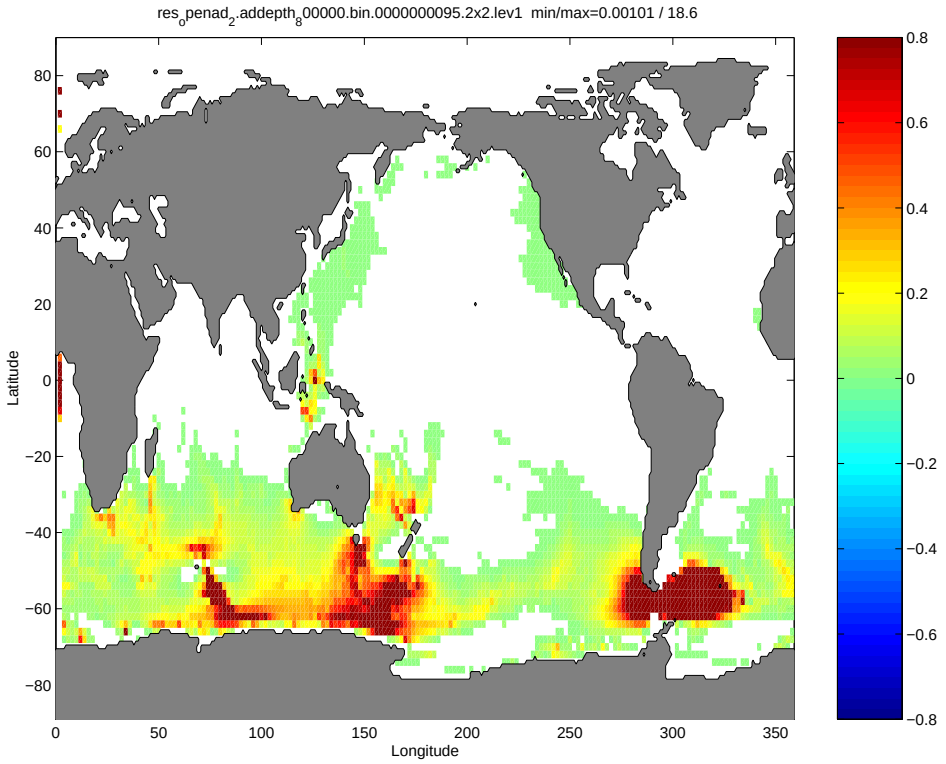
- modular design
- reusing existing components
- open source!
- language independence
- flexibility
- new AD algorithms
- did I mention open source?
- application to GCM code



OpenAD development

mechanics → tuning → mechanics → tuning → ...

- started out with small tests to verify numerics
- simple box model
- shallow water model (tuning via analysis and improved transformation)
- gcm configuration
  - mechanics sorted out
  - tuning to be refined



All of the above become part of a regression test set ensure some stability

OpenAD plans

relevant for this community:

- solidify/extend the Fortran front-end
- documented recipes for tool usage
- improved and new code analyses (activity, TBR, linearity)
- improved transformation (using heuristics and run-time profiles)
- efficient second order derivatives
- non-smoothness detection & handling in an optimization context

www.mcs.anl.gov/openad